

```
/* modo.c

Copyright (c) 1993-2012. Free Software Foundation, Inc.

This file is part of GNU MCSim.

GNU MCSim is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 3
of the License, or (at your option) any later version.

GNU MCSim is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with GNU MCSim; if not, see <http://www.gnu.org/licenses/>

-- Revisions -----
Logfile: %F%
Revision: %I%
Date: %G%
Modtime: %U%
Author: @a
-- SCCS -----

Output the model.c file.

Modified on 07/07/97 by FB - added yourcode.h to WriteIncludes().

*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>
#include <time.h>

#include "mod.h"
#include "lexfn.h"
#include "lexerr.h"
#include "modi.h"
#include "modd.h"
#include "modo.h"

/* Global Variables */

static char *vszModelFilename = NULL;
static char *vszModGenName = NULL;

static char vszModelArrayName[] = "vrgModelVars";
```

```

static char vszInputArrayName[] = "vrgInputs";
extern char vszHasInitializer[]; /* decl'd in modd.c */

PVMMAPSTRCT vpvmGloVarList;

char *vszIFNTypes[] = { /* Must match defines in lexfn.h */
    "IFN_NULL /* ?? */",
    "IFN_CONSTANT",
    "IFN_PERDOSE",
    "IFN_PERRATE",
    "IFN_PEREXP",
    "IFN_NDOSES"
}; /* vszIFNTypes[] = */

int vnStates, vnOutputs, vnInputs, vnParms, vnModelVars;

BOOL bForR = FALSE;
BOOL bForInits = FALSE;

/* -----
----- ForAllVar

Takes a pfile, a pvm list head and a callback function which is called
for all variables in the list if hType == ALL_VARS or only for
only those that match hType otherwise.
*/
int ForAllVar (PFILE pfile, PVMMAPSTRCT pvm, PFI_CALLBACK pfiFunc,
               HANDLE hType, PVOID pinfo)
{
    int iTotal = 0;

    while (pvm) {
        if (hType == ALL_VARS           /* Do for all ...*/
            || TYPE(pvm) == hType) {   /* ..or do only for vars of hType */
            if (pfiFunc)
                iTotal += (*pfiFunc) (pfile, pvm, pinfo);
            else
                iTotal++; /* No func! just count */
        }
        pvm = pvm->pvmNextVar;
    } /* while */

    return (iTotal);
} /* ForAllVar */

/* -----
----- CountOneDecl

Counts declarations for variables. Note that each variable may only

```

be declared and assigned *once*. This is a problem and should be fixed.

```
    Callback for ForAllVar().  
*/  
int CountOneDecl (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)  
{  
    if (pvm->szEqn != vszHasInitializer) { /* These appear later */  
        return (1);  
    }  
    return (0);  
} /* CountOneDecl */  
  
/* -----  
-----  
*/  
void WritebDelays (PFILE pfile, BOOL bDelays)  
{  
    fprintf (pfile, "\nBOOL bDelays = %d;\n", bDelays);  
} /* WritebDelays */  
  
/* -----  
-----  
WriteOneName  
  
Writes a name and nominal value for the given variable of pvm.  
  
    Callback for ForAllVar().  
*/  
int WriteOneName (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)  
{  
    if (pvm->szEqn != vszHasInitializer) { /* These appear later */  
        if (bForR) {  
            if (TYPE(pvm) == ID_OUTPUT)  
                fprintf (pfile, "    \"%s\", pvm->szName);  
            else  
                fprintf (pfile, "    %s", pvm->szName);  
  
            if (TYPE(pvm) != ID_INPUT) {  
                if (TYPE(pvm) == ID_OUTPUT)  
                    fprintf (pfile, "\\", \n");  
                else  
                    fprintf (pfile, " = %s,\n", (pvm->szEqn ? pvm->szEqn : "0.0"));  
            }  
            else  
                fprintf (pfile, " (forcing function)\n");  
        }  
        else { /* not for R */  
            fprintf (pfile, "    %s", pvm->szName);  
        }  
    }  
}
```

```

        if (TYPE(pvm) != ID_INPUT)
            fprintf (pfile, " %s %s;\n", (pvm->szEqn ? "=" : "->"),
                     (pvm->szEqn ? pvm->szEqn : "0.0"));
        else
            fprintf (pfile, " (is a function)\n");
    }
    return (1);
} /* if */

return (0);

} /* WriteOneName */

/*
-----
-----  

WriteHeader

That it does do dashingly. Includes a nice comment at the
beginning (hopefully!) of the file replete with filename,
timestamp, and a list of model variables.

*/
void WriteHeader (PFILE pfile, PSTR szName, PVMMAPSTRCT pvmGlo)
{
    time_t ttTime;

    time (&ttTime);

    if (fprintf (pfile, /* %s\n", szName) < 0)
        ReportError (NULL, RE_CANNOTOPEN | RE_FATAL,
                      szName, "...in WriteHeader ()");

    fprintf(pfile, "
                                         \n\n");
    fprintf (pfile, " Model File: %s\n\n", vszModelFilename);
    fprintf (pfile, " Date: %s\n", ctime(&ttTime));
    fprintf (pfile, " Created by: \"%s %s\"\n", vszModGenName,
VSZ_VERSION);
    fprintf (pfile, " -- a model preprocessor by Don Maszle\n");
    fprintf(pfile, "
                                         \n\n");
                                         \n\n");

    fprintf (pfile, " VSZ_COPYRIGHT "\n");

    fprintf (pfile, "\n    Model calculations for compartmental
model:\n\n");

    if (vnStates == 1) fprintf (pfile, "    1 State:\n");
    else fprintf (pfile, "    %d States:\n", vnStates);
    ForAllVar (pfile, pvmGlo, &WriteOneName, ID_STATE, NULL);

    if (vnOutputs == 1) fprintf (pfile, "\n    1 Output:\n");
    else fprintf (pfile, "\n    %d Outputs:\n", vnOutputs);
    ForAllVar (pfile, pvmGlo, &WriteOneName, ID_OUTPUT, NULL);
}

```

```

    if (vnInputs == 1) fprintf (pfile, "\n    1 Input:\n");
    else fprintf (pfile, "\n    %d Inputs:\n", vnInputs);
    ForAllVar (pfile, pvmGlo, &WriteOneName, ID_INPUT, NULL);

    if (vnParms == 1) fprintf (pfile, "\n    1 Parameter:\n");
    else fprintf (pfile, "\n    %d Parameters:\n", vnParms);
    ForAllVar (pfile, pvmGlo, &WriteOneName, ID_PARM, NULL);

    fprintf (pfile, "*/\n\n");

} /* WriteHeader */

/* -----
 */
void WriteIncludes (PFILE pfile)
{
    fprintf (pfile, "\n#include <stdlib.h>\n");
    fprintf (pfile, "#include <stdio.h>\n");
    fprintf (pfile, "#include <math.h>\n");
    fprintf (pfile, "#include <string.h>\n");
    fprintf (pfile, "#include <float.h>\n");

    fprintf (pfile, "#include \"modelu.h\"\n");
    fprintf (pfile, "#include \"random.h\"\n");
    fprintf (pfile, "#include \"yourcode.h\"\n");

} /* WriteIncludes */

/* -----
*/
WriteOneDecl

    Write one global or local declaration.  Callback for ForAllVar().

*/
int WriteOneDecl (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)
{
    assert (TYPE(pvm) != ID_INPUT);
    assert (TYPE(pvm) != ID_OUTPUT);
    assert (TYPE(pvm) != ID_STATE);

    if (TYPE(pvm) > ID_PARM) fprintf (pfile, "    /* local */ ");

    fprintf (pfile, "double %s;\n", pvm->szName);

    return (1);

} /* WriteOneDecl */

```

```

/* -----
*/
int WriteOneIndexDefine (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)
{
    if (pvm->szEqn != vszHasInitializer) { /* These appear later */
        fprintf (pfile, "#define ");
        WriteIndexName(pfile, pvm);
        if (INDEX(pvm))
            fprintf (pfile, " 0x%05lx\n", INDEX(pvm));
        else
            fprintf (pfile, " 0x00000\n");
    }

    return 1;
} /* if */

return 0;

} /* WriteOneIndexDefine */

/* -----
*/
void WriteDecls (PFILE pfile, PVMMAPSTRCT pvmGlo)
{
    fprintf (pfile, "\n\n/*---- Indices to Global Variables */\n");

    fprintf (pfile, "\n/* Model variables: States and other outputs */\n");
    ForAllVar (pfile, pvmGlo, &WriteOneIndexDefine, ID_STATE, NULL);
    ForAllVar (pfile, pvmGlo, &WriteOneIndexDefine, ID_OUTPUT, NULL);

    fprintf (pfile, "\n/* Inputs */\n");
    ForAllVar (pfile, pvmGlo, &WriteOneIndexDefine, ID_INPUT, NULL);
    fprintf (pfile, "\n/* Parameters */\n");
    ForAllVar (pfile, pvmGlo, &WriteOneIndexDefine, ID_PARM, NULL);

    fprintf (pfile, "\n\n/*---- Global Variables */\n");

    fprintf (pfile, "\n/* For export. Keep track of who we are. */\n");
    fprintf (pfile, "char szModelDescFilename[] = \"%s\";\n",
vszModelFilename);
    fprintf (pfile, "char szModelSourceFilename[] = FILE_;\n");
    fprintf (pfile, "char szModelGenAndVersion[] = \"%s %s\";\n",
vszModGenName, VSZ_VERSION);

    fprintf (pfile, "\n/* Externs */\n");
    fprintf (pfile, "extern BOOL vbModelReinitd;\n");

    fprintf (pfile, "\n/* Model Dimensions */\n");
    fprintf (pfile, "int vnStates = %d;\n", vnStates);
    fprintf (pfile, "int vnOutputs = %d;\n", vnOutputs);
    fprintf (pfile, "int vnModelVars = %d;\n", vnModelVars);
    fprintf (pfile, "int vnInputs = %d;\n", vnInputs);
}

```

```

fprintf (pfile, "int vnParms = %d;\n", vnParms);

fprintf (pfile, "\n/* States and Outputs*/\n");
fprintf (pfile, "double %s[%d];\n", vszModelArrayName, vnModelVars);

fprintf (pfile, "\n/* Inputs */\n");
/* if vnInputs is zero put a dummy 1 for array size */
fprintf (pfile, "IFN %s[%d];\n", vszInputArrayName,
        (vnInputs > 0 ? vnInputs : 1));

fprintf (pfile, "\n/* Parameters */\n");
ForAllVar (pfile, pvmGlo, &WriteOneDecl, ID_PARM, NULL);

} /* WriteDecls */

/*
-----
-----  

GetName

returns a string to the name of the pvm variable given. The
string is static and must be used immediately or copied. It will
be changed on the next call of this function.

szModelVarName and szDerivName are names to be used for
state variables and derivatives arrays, resp.

The name is determined by hType if hType is non-NULL. If hType is
NULL, then the type is taken to be the hType field of pvm.
*/
PSTR GetName (PVMMAPSTRCT pvm, PSTR szModelVarName, PSTR szDerivName,
              HANDLE hType)
{
    static PSTRLEX vszVarName;
    HANDLE hTypeToUse = (hType ? hType : TYPE(pvm));

    switch (hTypeToUse) {

    case ID_INPUT:
        if (bForR)
            sprintf (vszVarName, "%s", pvm->szName);
        else
            sprintf (vszVarName, "vrgInputs[ID_%s]", pvm->szName);
        break;

    case ID_STATE:
        if (bForR) {
            if (bForInits)
                sprintf (vszVarName, "%s", pvm->szName);
            else
                sprintf (vszVarName, "y[ID_%s]", pvm->szName);
        }
        else {
            if (szModelVarName)

```

```

        sprintf (vszVarName, "%s[ID_%s]", szModelVarName, pvm->szName);
    else
        sprintf (vszVarName, "vrgModelVars[ID_%s]", pvm->szName);
    }
    break;

case ID_OUTPUT:
    if (bForR)
        sprintf (vszVarName, "yout[ID_%s]", pvm->szName);
    else {
        if (szModelVarName)
            sprintf (vszVarName, "%s[ID_%s]", szModelVarName, pvm->szName);
        else
            sprintf (vszVarName, "vrgModelVars[ID_%s]", pvm->szName);
    }
    break;

case ID_DERIV:
    assert (szDerivName);
    if (bForR)
        sprintf (vszVarName, "ydot[ID_%s]", pvm->szName);
    else
        sprintf (vszVarName, "%s[ID_%s]", szDerivName, pvm->szName);
    break;

default: /* Params and local variables */
    sprintf (vszVarName, "%s", pvm->szName);
    break;
} /* switch */

return (vszVarName);

} /* GetName */

```

```

/*
-----
-----
WriteOneVMEntry

prints a single entry for the variable map symbol table.

Callback for ForAllList.
*/
int WriteOneVMEntry (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)
{
    int iType = TYPE(pvm);

    if (!pvm) {
        fprintf (pfile, " {\"\\\", NULL, 0} /* End flag */\n");
        return 0;
    }

    assert (iType && \

```

```

    iType != ID_LOCALDYN && iType != ID_LOCALSCALE && iType != ID_LOCALJACOB);

    if (pvm->szEqn != vszHasInitializer) { /* These appear later */
        fprintf (pfile, " {\\"%s\", (PVOID) &%s", pvm->szName,
                  GetName(pvm, vszModelArrayName, NULL, ID_NULL));

        fprintf (pfile, ", ID_%s | ID_%s},\n",
                (iType == ID_PARM ? "PARM"
                 : (iType == ID_INPUT ? "INPUT"
                   : (iType == ID_OUTPUT ? "OUTPUT"
                     : "STATE"))), pvm->szName);

        return (1);
    } /* if */

    return 0;
} /* WriteOneVMEntry */

/*
-----
*/
void WriteVarMap (PFILE pfile, PVMMAPSTRCT pvmGlo)
{
    fprintf (pfile, "\n\n/*---- Global Variable Map */\n\n");
    fprintf (pfile, "VMMAPSTRCT vrgvmGlo[] = {\n");
    ForAllVar (pfile, pvmGlo, &WriteOneVMEntry, ID_STATE, NULL);
    ForAllVar (pfile, pvmGlo, &WriteOneVMEntry, ID_OUTPUT, NULL);
    ForAllVar (pfile, pvmGlo, &WriteOneVMEntry, ID_INPUT, NULL);
    ForAllVar (pfile, pvmGlo, &WriteOneVMEntry, ID_PARM, NULL);
    WriteOneVMEntry (pfile, NULL, NULL); /* Include end flag */
    fprintf (pfile, "}; /* vrgpvmGlo[] */\n");

} /* WriteVarMap */

/*
-----
*/
int WriteOneInit (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)
{
    IFN_ifnNull = {IFN_CONSTANT}; /* Init other fields to zero */
    PSTR szVarName = GetName (pvm, NULL, NULL, ID_NULL);

    if ((pvm->szEqn != vszHasInitializer && /* These appear later */
         TYPE(pvm) <= ID_PARM) || /* No Locals! */
        (TYPE(pvm) == ID_INLINE)) {

        if (TYPE(pvm) == ID_INPUT) {
            PIFN pifn = (PIFN) pvm->szEqn;

```

```

    if (!pifn) /* No eqn, init to zero */
        pifn = &ifnNull;

    fprintf (pfile, " %s.iType = %s;\n",
             szVarName, vszIFNTypes[pifn->iType]);
    fprintf (pfile, " %s.dStartPeriod = 0;\n", szVarName);
    fprintf (pfile, " %s.bOn = FALSE;\n", szVarName);
    fprintf (pfile, " %s.dMag = %f;\n", szVarName, pifn-
>dMag);
    fprintf (pfile, " %s.dT0 = %f;\n", szVarName, pifn->dT0);
    fprintf (pfile, " %s.dTexp = %f;\n", szVarName, pifn-
>dTexp);
    fprintf (pfile, " %s.dDecay = %f;\n", szVarName, pifn-
>dDecay);
    fprintf (pfile, " %s.dTper = %f;\n", szVarName, pifn-
>dTper);

    fprintf (pfile, " %s.hMag = %#lx;\n", szVarName, pifn-
>hMag);
    fprintf (pfile, " %s.hT0 = %#lx;\n", szVarName, pifn->hT0);
    fprintf (pfile, " %s.hTexp = %#lx;\n", szVarName, pifn-
>hTexp);
    fprintf (pfile, " %s.hDecay = %#lx;\n", szVarName, pifn-
>hDecay);
    fprintf (pfile, " %s.hTper = %#lx;\n", szVarName, pifn-
>hTper);

    fprintf (pfile, " %s.dVal = 0.0;\n", szVarName);
    fprintf (pfile, " %s.nDoses = 0;\n", szVarName);
} /* if */
else {
    if (TYPE(pvm) == ID_INLINE) /* write out just the equation */
        fprintf (pfile, "\n%s\n", pvm->szEqn);
    else
        fprintf (pfile, " %s = %s;\n", szVarName,
                 (pvm->szEqn ? pvm->szEqn : "0.0"));
} /* else */

return (1);

} /* if */

return (0);

} /* WriteOneInit */

/*
-----
----- WriteInitModel

    Writes the routine to initialize the model variables.

*/
void WriteInitModel (PFILE pfile, PVMMAPSTRCT pvmGlo)

```

```

{
    fprintf (pfile, "\n\n/*---- InitModel\n");
    fprintf (pfile, "    Should be called to initialize model variables
at\n");
    fprintf (pfile, "    the beginning of experiment before reading\n");
    fprintf (pfile, "    variants from the simulation spec file.\n*/\n\n");
    fprintf (pfile, "void InitModel(void)\n{\n");
    fprintf (pfile, "    /* Initialize things in the order that they appear
in\n");
        "    model definition file so that dependencies are\n"
        "    handled correctly. */\n\n");
    ForAllVar (pfile, pvmGlo, &WriteOneInit, ALL_VARS, NULL);
    fprintf (pfile, "\n    vbModelReinitd = TRUE;\n\n");
    fprintf (pfile, "} /* InitModel */\n\n");

} /* WriteInitModel */

/*
-----
-----
TranslateID

    Writes an equation id if it declared and in a valid context.
    Errors are reported if necessary.
*/
void TranslateID (PINPUTBUF pibDum, PFILE pfile, PSTR szLex, int iEqType)
{
    int iKWCode, fContext;
    long iLowerB, iUpperB;

    iKWCode = GetKeywordCode (szLex, &fContext);

    switch (iKWCode) {

        case KM_DXDT: {
            int iArg = LX_IDENTIFIER;
            PVMMAPSTRCT pvm = NULL;

            if (GetFuncArgs (pibDum, 1, &iArg, szLex, &iLowerB, &iUpperB)
                && (pvm = GetVarPTR(vpvmGloVarList, szLex))
                && TYPE(pvm) == ID_STATE)
                fprintf (pfile, "%s", GetName (pvm, NULL, "rgDerivs", ID_DERIV));
            else
                ReportError (pibDum, RE_BADSTATE | RE_FATAL, (pvm? szLex : NULL),
NULL);
            } /* KM_DXDT block */
            break;

        case KM_NULL: {
            PVMMAPSTRCT pvm = GetVarPTR(vpvmGloVarList, szLex);

            /* Handle undeclared ids */
            if (!pvm) {

```

```

        if ((iEqType == KM_DYNAMICS ||  

            iEqType == KM_SCALE ||  

            iEqType == KM_CALCOUPUTS) &&  

            !(strcmp (szLex, VSZ_TIME) &&  

                strcmp (szLex, VSZ_TIME_SBML)))  

            /* If this is the time variable, convert to the correct formal  

arg */  

            fprintf(pfile, "(*pdTime)");  

        else  

            /* otherwise output id exactly as is */  

            fprintf(pfile, "%s", szLex);  

    } /* if */  
  

    else {  

        if (iEqType == KM_SCALE)  

            /* fixed by FB - 1 mar 98. This prints a "vrgModelVars" */  

            fprintf(pfile, "%s", GetName(pvm, NULL, NULL, ID_NULL));  

        else  

            fprintf(pfile, "%s", GetName(pvm, "rgModelVars", NULL, ID_NULL));  
  

        if ((TYPE(pvm) == ID_INPUT) && (!bForR))  

            fprintf(pfile, ".dVal"); /* Use current value */  

    } /* else */  
  

} /* KM_NULL: */  

break;  
  

default: /* No other keywords allowed in equations */  
  

/* Allow for C keywords here, including math library functions */  

ReportError (pibDum, RE_BADCONTEXT | RE_FATAL, szLex, NULL);  

break;  
  

} /* switch */  
  

} /* TranslateID */  
  

/* -----  
-----  

TranslateEquation  
  

Writes an equation to the output file substituting model variable  

names, inputs names, derivative names, etc.  
  

Tries to do some hack formatting.  

*/  
  

void TranslateEquation (PFILE pfile, PSTR szEqn, long iEqType)  

{  

#define RMARGIN 65  
  

    INPUTBUF ibDum;  

PINPUTBUF pibDum = &ibDum;

```

```

PSTRLEX szLex;
int iType;
BOOL bDelayCall = FALSE;

MakeStringBuffer (NULL, pibDum, szEqn);

NextLex (pibDum, szLex, &iType);
if (!iType) {
    fprintf (pfile, "0.0; /* NULL EQN!?? */");
    return;
} /* if */

do {
    if (iType == LX_IDENTIFIER) { // Process Identifier
        if (bDelayCall) {
            // do not translate the 1st param of CalcDelay but check it
            PVMMAPSTRCT pvm = NULL;
            if (((pvm = GetVarPTR(vpvmGloVarList, szLex)) &&
                (TYPE(pvm) == ID_STATE) || (TYPE(pvm) == ID_OUTPUT)) {
                fprintf (pfile, "ID_%s", szLex);
                // add the automatic time variable
                fprintf (pfile, ", (*pdTime)");
                bDelayCall = FALSE;
            }
            else
                ReportError (pibDum, RE_EXPECTED | RE_FATAL, "state or output",
NULL);
        }
        else
            TranslateID (pibDum, pfile, szLex, iEqType);
    }
    else {
        if ((iType == LX_EQNPUNCT || iType == LX_PUNCT) &&
            (*szLex) == CH_COMMENT)) {
            while (*pibDum->pbufCur && *pibDum->pbufCur != CH_EOLN)
                pibDum->pbufCur++;

            fprintf (pfile, "\n");
        }
        else /* Spew everything else */
            fprintf (pfile, "%s", szLex);
    }
}

if (!bDelayCall) // do not reset here if bDelayCall is TRUE
    bDelayCall = (!strcmp("CalcDelay", szLex));

fprintf (pfile, " ");
NextLex (pibDum, szLex, &iType);

} while (iType);

if (bForR && bForInits)
    fprintf (pfile, "\n");

```

```

    else
        fprintf (pfile, ";"\n");

} /* TranslateEquation */

/* -----
----- WriteOneEquation

Writes one equation of a function definition. Uses the hType
flag for spacing set by DefineEquation().

Callback function for ForAllVar().
*/
int WriteOneEquation (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)
{
    long iType = (long) pInfo;

    if (pvm->hType & ID_SPACEFLAG) /* Flag for space between equations */
        fprintf (pfile, "\n");

    switch (iType) {
        default:
        case KM_SCALE: /* Scale Global Names */

            /* Inputs not allowed anymore in Scale section - FB 7/12/96 */
            if (TYPE(pvm) == ID_INPUT) {
                printf ("Error: input '%s' used in Scale context.\n",
                       pvm->szName);
                exit(0);
            }

            if (TYPE(pvm) != ID_INLINE) { /* do not write "Inline" */
                if (bForR && bForInits && TYPE(pvm) == ID_STATE)
                    fprintf (pfile, " Y[\"%s\"] <- ",
                            GetName (pvm, NULL, NULL, ID_NULL));
                else
                    fprintf (pfile, " %s = ", GetName (pvm, NULL, NULL, ID_NULL));
            }
            break;

        case KM_CALCOUTPUTS:
        case KM_DYNAMICS:
        case KM_JACOB:
        case KM_EVENTS:
        case KM_ROOTS:
            if (TYPE(pvm) != ID_INLINE) /* do not write "Inline" */
                fprintf (pfile, " %s = ", GetName (pvm, "rgModelVars",
"rgDerivs",
                           ID_NULL));
            break;
    } /* switch */
}

```

```

if (TYPE(pvm) == ID_INLINE) /* write out the equation */
    fprintf (pfile, "\n%s\n", pvm->szEqn);
else
    TranslateEquation (pfile, pvm->szEqn, iType);

return 1;

} /* WriteOneEquation */

/*
-----
----- WriteCalcDeriv

Writes the CalcDeriv() function. Writes dynamics equations in
the order they appeared in the model definition file.

*/
void WriteCalcDeriv (PFILE pfile, PVMMAPSTRCT pvmGlo, PVMMAPSTRCT pvmDyn)
{
    if (!pvmDyn)
        printf ("No Dynamics{} equations.\n\n");

    fprintf (pfile, "/*---- Dynamics section */\n\n");
    fprintf (pfile, "void CalcDeriv (double rgModelVars[], ");
    fprintf (pfile, "double rgDerivs[], PDOUBLE pdTime)\n{\n");

    ForAllVar (pfile, pvmGlo, &WriteOneDecl, ID_LOCALDYN, NULL);

    fprintf (pfile, "\n    CalcInputs (pdTime); /* Get new input vals
*/\n\n");

    ForAllVar (pfile, pvmDyn, &WriteOneEquation, ALL_VARS, (PVOID)
KM_DYNAMICS);

    fprintf (pfile, "\n} /* CalcDeriv */\n\n\n");

} /* WriteCalcDeriv */

/*
-----
----- WriteScale

*/
void WriteScale (PFILE pfile, PVMMAPSTRCT pvmGlo, PVMMAPSTRCT pvmScale)
{
    if (!pvmScale)
        printf ("No Scale{} equations. Null function defined.\n\n");

    fprintf (pfile, "/*---- Model scaling */\n\n");
    fprintf (pfile, "void ScaleModel (PDOUBLE pdTime)\n");
    fprintf (pfile, "{\n");
}

```

```

    ForAllVar (pfile, pvmGlo, &WriteOneDecl, ID_LOCALSCALE, NULL);
    ForAllVar (pfile, pvmScale, &WriteOneEquation, ALL_VARS, (PVOID)
KM_SCALE);
    fprintf (pfile, "\n} /* ScaleModel */\n\n\n");

} /* WriteScale */

/*
-----
----- WriteCalcJacob
*/
void WriteCalcJacob (PFILE pfile, PVMMAPSTRCT pvmGlo, PVMMAPSTRCT
pvmJacob)
{
/* if (!pvmJacob)
   printf ("No Jacobian{} equations. Null function defined.\n\n"); */

fprintf (pfile, "/*---- Jacobian calculations */\n\n");
fprintf (pfile, "void CalcJacob (PDOUBLE pdTime, double
rgModelVars[],\n");
fprintf (pfile, "           long column, double rgdJac[])\n");
fprintf (pfile, "{\n");
ForAllVar (pfile, pvmGlo, &WriteOneDecl, ID_LOCALJACOB, NULL);
ForAllVar (pfile, pvmJacob, &WriteOneEquation, ALL_VARS, (PVOID)
KM_JACOB);
fprintf (pfile, "\n} /* CalcJacob */\n\n\n");

} /* WriteCalcJacob */

/*
-----
----- WriteCalcOutputs
*/
void WriteCalcOutputs (PFILE pfile, PVMMAPSTRCT pvmGlo, PVMMAPSTRCT
pvmCalcOut)
{
if (!pvmCalcOut)
   printf ("No CalcOutputs{} equations. Null function defined.\n\n");

fprintf (pfile, "/*---- Outputs calculations */\n\n");
fprintf (pfile, "void CalcOutputs (double rgModelVars[], ");
fprintf (pfile, "double rgDerivs[], PDOUBLE pdTime)\n{\n");
ForAllVar (pfile, pvmGlo, &WriteOneDecl, ID_LOCALCALCOUT, NULL);
ForAllVar (pfile, pvmCalcOut, &WriteOneEquation, ALL_VARS,
(PVOID) KM_CALCOUPUTS);
fprintf (pfile, "\n} /* CalcOutputs */\n\n\n");

} /* WriteCalcOutputs */

/*
-----
-----
```

IndexOneVar

ORs the index passed through the info pointer, a PINT, and increments the value of the pointer.

The handle is corrected for use in the simulation. For this, the TYPE part of the handle is shifted left 4 bits.

Callback function for ForAllVar().

```
/*
int IndexOneVar (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)
{
    if (pvm->szEqn != vszHasInitializer) { /* Handled later */
        pvm->hType |= (*((PINT) pInfo))++;
        return 1;
    } /* if */
    return 0;
} /* IndexOneVar */
```

```
/* -----
-----
```

IndexVariables

Creates indices and counts all the model variables.

The model variables (states and outputs) are indexed together, since they will be stored in one array.

The indices to the parameters are into the global var map.

The indices to inputs are into the array of inputs.

```
/*
void IndexVariables (PVMMAPSTRCT pvmGlo)
{
    int iIndex, iMax = MAX_VARS;

    /* Get counts */
    vnStates = ForAllVar (NULL, pvmGlo, &CountOneDecl, ID_STATE, NULL);
    vnOutputs = ForAllVar (NULL, pvmGlo, &CountOneDecl, ID_OUTPUT, NULL);
    vnInputs = ForAllVar (NULL, pvmGlo, &CountOneDecl, ID_INPUT, NULL);
    vnParms = ForAllVar (NULL, pvmGlo, &CountOneDecl, ID_PARM, NULL);
    vnModelVars = vnStates + vnOutputs;

    /* Report all errors */
    if (vnStates > MAX_VARS)
        ReportError (NULL, RE_TOOMANYVARS, "state", (PSTR) &iMax);
    if (vnOutputs > MAX_VARS)
        ReportError (NULL, RE_TOOMANYVARS, "input", (PSTR) &iMax);
    if (vnInputs > MAX_VARS)
        ReportError (NULL, RE_TOOMANYVARS, "output", (PSTR) &iMax);
    if (vnParms > (iMax = MAX_VARS - vnModelVars))
        ReportError (NULL, RE_TOOMANYVARS, "parameter", (PSTR) &iMax);
```

```

if (vnStates > MAX_VARS
    || vnInputs > MAX_VARS
    || vnOutputs > MAX_VARS
    || vnParms > iMax)
    ReportError (NULL, RE_FATAL, NULL, NULL); /* Abort generation */

/* Set indices */
iIndex = 0;
ForAllVar (NULL, pvmGlo, &IndexOneVar, ID_STATE, (PVOID) &iIndex);
ForAllVar (NULL, pvmGlo, &IndexOneVar, ID_OUTPUT, (PVOID) &iIndex);

iIndex = 0;
ForAllVar (NULL, pvmGlo, &IndexOneVar, ID_INPUT, (PVOID) &iIndex);

iIndex = vnStates + vnOutputs + vnInputs;
ForAllVar (NULL, pvmGlo, &IndexOneVar, ID_PARM, (PVOID) &iIndex);

} /* IndexVariables */

/*
-----
----- AdjustOneVar

Increments the dependent parameter handles of an input by the
iOffset given through the info pointer.

Callback function for ForAllVar().

*/
int AdjustOneVar (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)
{
    PIFN pifn = (PIFN) pvm->szEqn;
    WORD wOffset = *(PWORD) pInfo;

    if (!pifn)
        return 1; /* No eqn! No dependent parm! */

    if (pifn->hMag)
        pifn->hMag += wOffset;
    if (pifn->hTper)
        pifn->hTper += wOffset;
    if (pifn->hT0)
        pifn->hT0 += wOffset;
    if (pifn->hTexp)
        pifn->hTexp += wOffset;
    if (pifn->hDecay)
        pifn->hDecay += wOffset;

    return 1;
} /* AdjustOneVar */

```

```

/* -----
----- AdjustVarHandles

Adjusts the variable handles on input definitions. They must be
incremented by the beginning offset of the parameter section of
the global variable map.
*/
void AdjustVarHandles (PVMMAPSTRCT pvmGlo)
{
    WORD wOffset = (WORD) vnInputs + vnStates + vnOutputs;

    ForAllVar (NULL, pvmGlo, &AdjustOneVar, ID_INPUT, (PVOID) &wOffset);

} /* AdjustVarHandles */

/* -----
----- ReversePointers

Flips the pointer on the var-list so that the head is the tail
and the tail is the head and things aren't what they seem to be.

The dynamic equation list must be reversed so that the equations
appear in the right order since they were created as a stack.
*/
void ReversePointers (PVMMAPSTRCT *ppvm)
{
    PVMMAPSTRCT pvmPrev, pvmNext;

    if (!ppvm || !(*ppvm)
        || !(*ppvm)->pvmNextVar) /* List of one is already reversed! */
        return;

    pvmPrev = NULL;
    while ((pvmNext = (*ppvm)->pvmNextVar)) {
        (*ppvm)->pvmNextVar = pvmPrev;
        pvmPrev = (*ppvm);
        *ppvm = pvmNext;
    } /* while */

    (*ppvm)->pvmNextVar = pvmPrev; /* Link new head to reversed list */

} /* ReversePointers */

/* -----
----- AssertExistsEqn

Check that equation exists.

Uses info pointer of ForAllVar's callback as a second eqn list.

```

(1) If the second list is NULL, checks for initialization of pvm.

(2) If the second list is non-NULL, checks it for definition.

Note that case (1) and (2) apply to inputs and dynamics eqns respectively.

Further assertion will have to be handled otherwise. parms are assured to be initialized when declared, and outputs can be initialized to 0 automatically.

The errors are not reported as fatal so that all errors can be discovered. They will cause exit subsequently.

* /

```
int AssertExistsEqn (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)
```

{

```
int iReturn = 0;  
PVMMAPSTRCT pvmDyn = (PVMMAPSTRCT) pInfo;
```

```

if (pvm->szEqn != vszHasInitializer) { /* Don't count these! */
    if (pvmDyn) {
        if (!(iReturn = (GetVarPTR (pvmDyn, pvm->szName) != NULL)))
            ReportError (NULL, RE_NODYNEQN, pvm->szName, NULL);
    }
    else
        if (!(iReturn = (pvm->szEqn != NULL)))
            ReportError (NULL, RE_NOINPDEF, pvm->szName, NULL);
} /* if */

```

```
/* If HasInitializer, no error to report */
return (iReturn ? 1 : 0);
```

```
} /* AssertExistsEqn */
```

```
/* -----  
-----
```

Calls `AssertExistsEqn` on each and return a fatal error if one or more equation is missing.

1 /

```
void VerifyEans (PVMMAPSTRUCT numGlo, PVMMAPSTRUCT numDyn)
```

1

BOOT: hStatesOK:

```
bStatesOK = (vnStates == ForAllVar (NULL, pvmGlo, &AssertExistsEqn,  
        ID STATE, (PVOID) pvmDyn));
```

```

    if (!bStatesOK)
        ReportError (NULL, RE_FATAL, NULL, "State equations missing.\n");

} /* VerifyEqns */


/* -----
-----
AssertExistsOutputEqn

Check that an equation exists in either the Dynamics section or the
CalcOutputs section for each output variable.

The errors are not reported as fatal so that all errors can
be discovered. They will cause exit subsequently.
*/
int AssertExistsOutputEqn (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)
{
    int iReturn = 0;
    PINPUTINFO pinfo = (PINPUTINFO) pInfo;
    PVMMAPSTRCT pvmDyn = (PVMMAPSTRCT) pinfo->pvmDynEqns;
    PVMMAPSTRCT pvmOut = (PVMMAPSTRCT) pinfo->pvmCalcOutEqns;

    if (pvm->szEqn != vszHasInitializer) { /* Don't count these! */

        if (((GetVarPTR (pvmDyn, pvm->szName) == NULL) &&
            (GetVarPTR (pvmOut, pvm->szName) == NULL)) {
            ReportError (NULL, RE_NOOUTPUTEQN, pvm->szName, NULL);
            iReturn = 0;
        }
        else
            iReturn = 1;
    } /* if */

    /* If HasInitializer, no error to report */
    return (iReturn ? 1 : 0);
}

} /* AssertExistsOutputEqn */


/* -----
-----
VerifyOutputEqns

Check that each output variable has an associated equation defined.

Calls AssertExistsOutputEqn on each and return a fatal error if one or
more
equation is missing.
*/
void VerifyOutputEqns (PINPUTINFO pInfo)
{
    BOOL bOutputsOK;

```

```

bOutputsOK = (vnOutputs == ForAllVar (NULL, pInfo->pvmGloVars,
                                      &AssertExistsOutputEqn,
                                      ID_OUTPUT,
                                      (PVOID) pInfo));
if (!bOutputsOK)
    ReportError (NULL, RE_FATAL, NULL, "Output equations missing.\n");
} /* VerifyOutputEqns */

/* -----
----- WriteModel

    Writes the model calculation file szOutFilename for the parameters
    and dynamic equations given.

*/
void WriteModel (PINPUTINFO pInfo, PSTR szFileOut)
{
    PFILE pfile;

    if (!pInfo->pvmGloVars || (!pInfo->pvmDynEqns && !pInfo-
>pvmCalcOutEqns)) {
        printf ("Error: No Dynamics, no outputs or no global variables
defined\n");
        return;
    }

    ReversePointers (&pInfo->pvmGloVars);
    ReversePointers (&pInfo->pvmDynEqns);
    ReversePointers (&pInfo->pvmScaleEqns);
    ReversePointers (&pInfo->pvmCalcOutEqns);
    ReversePointers (&pInfo->pvmJacobEqns);
    vPvmGloVarList = pInfo->pvmGloVars;

    IndexVariables (pInfo->pvmGloVars);
    AdjustVarHandles (pInfo->pvmGloVars);
    VerifyEqns (pInfo->pvmGloVars, pInfo->pvmDynEqns);

    VerifyOutputEqns (pInfo);

    pfile = fopen (szFileOut, "w");
    if (pfile) {

        /* Keep track of the model description file and generator name */
        vszModelFilename = pInfo->szInputFilename;
        vszModGenName = pInfo->szModGenName;

        WriteHeader (pfile, szFileOut, pInfo->pvmGloVars);

        WriteIncludes (pfile);
        WriteDecls (pfile, pInfo->pvmGloVars);
        WritebDelays (pfile, pInfo->bDelays);
    }
}

```

```

        WriteVarMap    (pfile, pinfo->pvmGloVars);

        WriteInitModel   (pfile, pinfo->pvmGloVars);
        WriteCalcDeriv   (pfile, pinfo->pvmGloVars, pinfo->pvmDynEqns);
        WriteScale        (pfile, pinfo->pvmGloVars, pinfo->pvmScaleEqns);
        WriteCalcJacob   (pfile, pinfo->pvmGloVars, pinfo->pvmJacobEqns);
        WriteCalcOutputs  (pfile, pinfo->pvmGloVars, pinfo->pvmCalcOutEqns);

        fclose (pfile);

        printf ("\n* Created model file '%s'.\n\n", szFileOut);

    } /* if */
else
    ReportError (NULL, RE_CANNOTOPEN | RE_FATAL,
                  szFileOut, "...in WriteModel ()");

} /* WriteModel */

/*
-----
*/
int WriteOne_R_SODefine (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)
{
    static long iStates = 0; /* indexing states */
    static long iOutputs = 0; /* indexing outputs */

    if (pvm->szEqn != vszHasInitializer) {
        fprintf (pfile, "#define ");
        WriteIndexName(pfile, pvm);
        if (TYPE(pvm) == ID_STATE) {
            fprintf (pfile, " 0x%05lx\n", iStates);
            iStates = iStates + 1;
        }
        else {
            fprintf (pfile, " 0x%05lx\n", iOutputs);
            iOutputs = iOutputs + 1;
        }
        return 1;
    }

    return 0;
}

} /* WriteOne_R_SODefine */

/*
-----
Write_R_Scale
*/
void Write_R_Scale (PFILE pfile, PVMMAPSTRCT pvmGlo, PVMMAPSTRCT
pvmScale)

```

```

{
    fprintf (pfile,
              "void getParms (double *inParms, double *out, int *nout)
{\n");
    fprintf (pfile, "/*---- Model scaling */\n\n");

    ForAllVar (pfile, pvmGlo, &WriteOneDecl, ID_LOCALSCALE, NULL);

    fprintf (pfile, "    int i;\n\n");
    fprintf (pfile, "    for (i = 0; i < *nout; i++) {\n");
    fprintf (pfile, "        parms[i] = inParms[i];\n    }\n\n");

    ForAllVar (pfile, pvmScale, &WriteOneEquation, ID_PARM, (PVOID)
KM_SCALE);

    fprintf (pfile, "\n    for (i = 0; i < *nout; i++) {\n");
    fprintf (pfile, "        out[i] = parms[i];\n    }\n");
    fprintf (pfile, "}\n");

} /* Write_R_Scale */

/*
-----
----- Write_R_State_Scale
*/
void Write_R_State_Scale (PFILE pfile, PVMMAPSTRCT pvmScale)
{

    ForAllVar (pfile, pvmScale, &WriteOneEquation, ID_STATE, (PVOID)
KM_SCALE);

    ForAllVar (pfile, pvmScale, &WriteOneEquation, ID_INLINE, (PVOID)
KM_SCALE);

} /* Write_R_State_Scale */

/*
-----
----- Write_R_CalcDeriv

    Writes the CalcDeriv() function for compatibility with R deSolve
package.
    Writes dynamics equations in the order they appeared in the model
definition
    file. Appends the CalcOutput equations, if any, at the end.
*/
void Write_R_CalcDeriv (PFILE pfile, PVMMAPSTRCT pvmGlo, PVMMAPSTRCT
pvmDyn,
                        PVMMAPSTRCT pvmCalcOut)
{
    if (!pvmDyn)
        printf ("No Dynamics{} equations.\n\n");
}

```

```

fprintf (pfile, "----- Dynamics section */\n\n");
fprintf (pfile, "void derivs (int *neq, double *pdTime, double *y, ");
fprintf (pfile, "double *ydot, double *yout, int *ip)\n{\n");

ForAllVar (pfile, pvmGlo, &WriteOneDecl, ID_LOCALDYN, NULL);
ForAllVar (pfile, pvmGlo, &WriteOneDecl, ID_LOCALCALCOUT, NULL);

ForAllVar (pfile, pvmDyn, &WriteOneEquation, ALL_VARS, (PVOID)
KM_DYNAMICS);

ForAllVar (pfile, pvmCalcOut, &WriteOneEquation, ALL_VARS,
(PVOID) KM_CALCOUPUTS);

fprintf (pfile, "\n} /* derivs */\n\n\n");

} /* Write_R_CalcDeriv */

/*
-----
----- Write_R_InitModel

    Writes the routine to initialize the model variables for R deSolve.
*/
void Write_R_InitModel (PFILE pfile, PVMMAPSTRCT pvmGlo)
{
    fprintf (pfile, "----- Initializers */\n");
    fprintf (pfile, "void initmod (void (*odeparms)(int *, double
*))\n{\n");
    fprintf (pfile, "    int N=%d;\n", vnParms);
    fprintf (pfile, "    odeps(&N, parms);\n");
    fprintf (pfile, "}\n\n");

    fprintf (pfile, "void initforc (void (*odeforcs)(int *, double
*))\n{\n");
    fprintf (pfile, "    int N=%d;\n", vnInputs);
    fprintf (pfile, "    odeforcs(&N, forc);\n");
    fprintf (pfile, "}\n\n\n");

} /* Write_R_InitModel */

/*
-----
----- Write_R_CalcJacob: empty in fact
*/
void Write_R_CalcJacob (PFILE pfile, PVMMAPSTRCT pvmGlo, PVMMAPSTRCT
pvmJacob)
{
    fprintf (pfile, "----- Jacobian calculations: */\n");
    fprintf (pfile, "void jac (int *neq, double *t, double *y, int *ml, ");
    fprintf (pfile, "int *mu, ");
    fprintf (pfile, "double *pd, int *nrowpd, double *yout, int *ip)\n");
}

```

```

fprintf (pfile, "{\n");
ForAllVar (pfile, pvmGlo, &WriteOneDecl, ID_LOCALJACOB, NULL);
ForAllVar (pfile, pvmJacob, &WriteOneEquation, ALL_VARS, (PVOID)
KM_JACOB);
fprintf (pfile, "\n} /* jac */\n\n\n");

} /* Write_R_CalcJacob */

/* -----
-----
      Write_R_Events: may contain Inlines
*/
void Write_R_Events (PFILE pfile, PVMMAPSTRCT pvmGlo, PVMMAPSTRCT
pvmEvents)
{
    fprintf (pfile, "/*---- Events calculations: */\n");
    fprintf (pfile, "void event (int *n, double *t, double *y)\n");
    fprintf (pfile, "{\n");
    ForAllVar (pfile, pvmGlo, &WriteOneDecl, ID_LOCALEVENT, NULL);
    ForAllVar (pfile, pvmEvents, &WriteOneEquation, ALL_VARS, (PVOID)
KM_EVENTS);
    fprintf (pfile, "\n} /* event */\n\n");

} /* Write_R_Events */

/* -----
-----
      Write_R_Roots: may contain Inlines
*/
void Write_R_Roots (PFILE pfile, PVMMAPSTRCT pvmGlo, PVMMAPSTRCT
pvmRoots)
{
    fprintf (pfile, "/*---- Roots calculations: */\n");
    fprintf (pfile, "void root (int *neq, double *t, double *y, ");
    fprintf (pfile, "int *ng, double *gout, double *out, int *ip)\n");
    fprintf (pfile, "{\n");
    ForAllVar (pfile, pvmGlo, &WriteOneDecl, ID_LOCALROOT, NULL);
    ForAllVar (pfile, pvmRoots, &WriteOneEquation, ALL_VARS, (PVOID)
KM_ROOTS);
    fprintf (pfile, "\n} /* root */\n\n");

} /* Write_R_Roots */

/* -----
-----
      WriteOne_R_PIDefine

      Write one global or local declaration for parameters and inputs
(forcing
functions) passed through R.
Increments and prints two parallel counters ("iParms" and "iForcs").

```

```

    Callback for ForAllVar() .
*/
int WriteOne_R_PIDefine (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)
{
    static long iParms = 0; /* indexing parameters */
    static long iForcs = 0; /* indexing input functions (forcing function)
*/
    assert (TYPE(pvm) != ID_OUTPUT);
    assert (TYPE(pvm) != ID_STATE);

    if (TYPE(pvm) == ID_PARM) {
        fprintf (pfile, "#define %s parms[%ld]\n", pvm->szName, iParms);
        iParms = iParms + 1;
    }
    else {
        fprintf (pfile, "#define %s forc[%ld]\n", pvm->szName, iForcs);
        iForcs = iForcs + 1;
    }

    return (1);
} /* WriteOne_R_PIDefine */

```

```

/* -----
-----
ForAllVarwSep

Takes a pfile, a pvm list head and a callback function which is called
for all variables in the list if hType == ALL_VARS or only for
only those that match hType otherwise. This version is for writing
lists with separating punctuation. It sets End to -1 for the first
item in the list, 0 for items in the middle of the list, and 1 for the
last item.
*/
int ForAllVarwSep (PFILE pfile, PVMMAPSTRCT pvm, PFI_CALLBACK pfiFunc,
                   HANDLE hType, PVOID pinfo)
{
    int iTotal = 0;
    long End = -1;
    int iCount = 0;

    while (pvm) {
        if (hType == ALL_VARS           /* Do for all ... */
            || TYPE(pvm) == hType) {   /* ... or do only for vars of hType */

            if (pvm->szEqn != vszHasInitializer) {
                if (pfiFunc) {
                    if (iCount > 0)
                        End = 0;
                    iTotal += (*pfiFunc) (pfile, pvm, (PVOID) End);
                    iCount++;
                }
            }
        }
    }
}
```

```

        else
            iTotal++; /* No func! just count */
    }
}
pvm = pvm->pvmNextVar;
} /* while */

End = 1;

(*pfiFunc) (pfile, pvm, (PVOID) End);

return (iTotal);

} /* ForAllVarwSep */

/* -----
-----
WriteOne_R_ParmDecl

Write an R (comma-separated list) of parameter or state names. pInfo
is
used as a beginning - middle -end list flag.
*/
int WriteOne_R_PSDecl (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)
{
PSTR szVarName;
long End = (long) pInfo;

if (End < 1)
    szVarName = GetName (pvm, NULL, NULL, ID_NULL);

switch (End) {
case -1:
    fprintf (pfile, "      %s = 0.0", szVarName);
    break;

case 0:
    fprintf (pfile, ",\n      %s = 0.0", szVarName);
    break;

case 1:
    fprintf (pfile, "\n");
    return (0);
}

return (1);

} /* WriteOne_R_PSDecl */

/* -----
-----
WriteOne_R_ParmInit

```

```

/*
int WriteOne_R_ParmInit (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)
{
    PSTR szVarName;

    if (((long) pInfo) < 1) {
        szVarName = GetName (pvm, NULL, NULL, ID_NULL);

        fprintf (pfile, "    %s = %s;\n", szVarName,
                 (pvm->szEqn ? pvm->szEqn : "0.0"));

        return (1);
    }
    else
        return (0);
}

} /* WriteOne_R_ParmInit */

/* -----
-----
    WriteOne_R_StateInit
*/
int WriteOne_R_StateInit (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)
{
    PSTR szVarName;
    long End = (long) pInfo;

    if (End < 1)
        szVarName = GetName (pvm, NULL, NULL, ID_NULL);

    switch (End) {
        case -1:
            fprintf (pfile, "    %s = %s", szVarName,
                     (pvm->szEqn ? pvm->szEqn : "0.0"));
            break;

        case 0:
            fprintf (pfile, ",\n    %s = %s", szVarName,
                     (pvm->szEqn ? pvm->szEqn : "0.0"));
            break;

        case 1:
            fprintf (pfile, "\n");
            return (0);
    }

    return (1);
}

} /* WriteOne_R_StateInit */

/* -----
-----

```

```

    WriteOneOutputName

        For R only
*/
int WriteOneOutputName (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)
{
    long END = (long) pInfo;

    switch (END) {
        case -1:
            fprintf (pfile, "      \"%s\\\"", pvm->szName);
            break;

        case 0:
            fprintf (pfile, ",\\n      \"%s\\\"", pvm->szName);
            break;

        case 1:
            fprintf (pfile, "\\n");
    }

    return (1);
} /* WriteOneOutputName */

/*
-----
----- This is where we are going...

int WriteOneRForcing (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)
{
    PIFN pifn = (PIFN) pvm->szEqn;
}

void Write_R_Forcings(PFILE pfile, PVMMAPSTRCT pvmGlo)
{
    ForAllVar (pfile, pvmGlo, &WriteOneRForcing, ID_INIT, NULL);
}
*/

/*
-----
----- Write_R_InitPOS

    Write a utility R file defining functions to initialize parameters,
outputs
    and states.
*/
void Write_R_InitPOS (PFILE pfile, PVMMAPSTRCT pvmGlo, PVMMAPSTRCT
pvmScale)
{
    /* write R function initParms */

```

```

fprintf (pfile, "initParms <- function(newParms = NULL) {\n");
fprintf (pfile, "  parms <- c(\n");
ForAllVarwSep (pfile, pvmGlo, &WriteOne_R_PSDecl, ID_PARM, NULL);
fprintf (pfile, " )\n");
fprintf (pfile, "  parms <- within(as.list(parms), {\n");
ForAllVarwSep (pfile, pvmGlo, &WriteOne_R_ParmInit, ID_PARM, NULL);
fprintf (pfile, " })\n");
fprintf (pfile, "  if (!is.null(newParms)) {\n");
fprintf (pfile, "    if (!all(names(newParms) %%in%% c(names(parms))))\n");
{\n";
fprintf (pfile, "      stop(\"illegal parameter name\")\n");
fprintf (pfile, "    }\n");
fprintf (pfile, "  }\n");
fprintf (pfile, "  if (!is.null(newParms))\n");
fprintf (pfile, "    parms[names(newParms)] <- newParms\n");
fprintf (pfile, "    out <- .C(\"getParms\", as.double(parms),\n");
fprintf (pfile, "      out=double(length(parms)),\n");
fprintf (pfile, "      as.integer(length(parms)))$out\n");
fprintf (pfile, "    names(out) <- names(parms)\n");
fprintf (pfile, "    out\n");
fprintf (pfile, "}\n\n");

/* write R function Outputs */
fprintf (pfile, "Outputs <- c(\n");
ForAllVarwSep (pfile, pvmGlo, &WriteOneOutputName, ID_OUTPUT, NULL);
fprintf (pfile, ")\n\n");

/* write R function initStates */
bForInits = TRUE;
fprintf (pfile, "initStates <- function(parms, newState = NULL)"
" {\n  Y <- c(\n");
ForAllVarwSep (pfile, pvmGlo, &WriteOne_R_PSDecl, ID_STATE, NULL);
fprintf (pfile, " )\n");

// do the next only if needed otherwise Y is reset to null
if (ForAllVar (pfile, pvmScale, NULL, ID_STATE, NULL) ||
    ForAllVar (pfile, pvmScale, NULL, ID_INLINE, NULL)) {
    fprintf (pfile, "  Y <- within(as.list(parms), {"));
    Write_R_State_Scale (pfile, pvmScale);
    fprintf (pfile, "\n  })$Y\n\n");
}

fprintf (pfile, "  if (!is.null(newState)) {\n");
fprintf (pfile, "    if (!all(names(newState) %%in%% c(names(Y))))\n");
{\n";
    fprintf (pfile,
"      stop(\"illegal state variable name in newState\")\n");
    fprintf (pfile, "    }\n");
    fprintf (pfile, "    Y[names(newState)] <- newState\n  }\n  Y\n}\n");

/* Write_R_Forcings: unfinished;
   translate mcsim's input information into a default forcing
   formulation for deSolve solvers. */

```

```

bForInits = FALSE;

} /* Write_R_InitPOS */

/*
-----
----- Write_R_Decls -----
*/
void Write_R_Decls (PFILE pfile, PVMMAPSTRCT pvmGlo)
{
    fprintf (pfile, "\n/* Model variables: States */\n");
    ForAllVar (pfile, pvmGlo, &WriteOne_R_SODefine, ID_STATE, NULL);

    fprintf (pfile, "\n/* Model variables: Outputs */\n");
    ForAllVar (pfile, pvmGlo, &WriteOne_R_SODefine, ID_OUTPUT, NULL);

    fprintf (pfile, "\n/* Parameters */\n");
    fprintf (pfile, "static double parms[%d];\n\n", vnParms);
    ForAllVar (pfile, pvmGlo, &WriteOne_R_PIDefine, ID_PARM, NULL);

    fprintf (pfile, "\n/* Forcing (Input) functions */\n");
    fprintf (pfile, "static double forc[%d];\n\n", vnInputs);
    ForAllVar (pfile, pvmGlo, &WriteOne_R_PIDefine, ID_INPUT, NULL);
    fprintf (pfile, "\n");

} /* Write_R_Decls */

/*
-----
----- Write_R_Includes -----
*/
void Write_R_Includes (PFILE pfile)
{
    fprintf (pfile, "#include <R.h>\n");
} /* Write_R_Includes */

/*
-----
----- Write_R_Model
Writes a deSolve (R package) compatible C file "szOutFilename"
corresponding to equations given.
*/
void Write_R_Model (PINPUTINFO pinfo, PSTR szFileOut)
{
    static PSTRLEX vszModified_Title;
    PFILE pfile;
    PSTR Rfile;
    PSTR Rappend = "_inits.R";
    size_t nRout, nbase;
    char * lastdot;
}

```

```

/* set global flag ! */
bForR = TRUE;

if (!pinfo->pvmGloVars || (!pinfo->pvmDynEqns && !pinfo-
>pvmCalcOutEqns)) {
    printf ("Error: No Dynamics, outputs or global variables defined\n");
    return;
}

ReversePointers (&pinfo->pvmGloVars);
ReversePointers (&pinfo->pvmDynEqns);
ReversePointers (&pinfo->pvmScaleEqns);
ReversePointers (&pinfo->pvmCalcOutEqns);
ReversePointers (&pinfo->pvmJacobEqns);
ReversePointers (&pinfo->pvmEventEqns);
ReversePointers (&pinfo->pvmRootEqns);
vpvmGloVarList = pinfo->pvmGloVars;

IndexVariables (pinfo->pvmGloVars);
AdjustVarHandles (pinfo->pvmGloVars);
VerifyEqns (pinfo->pvmGloVars, pinfo->pvmDynEqns);

VerifyOutputEqns (pinfo);

pfile = fopen (szFileOut, "w");
if (pfile) {

    /* Keep track of the model description file and generator name */
    vszModelFilename = pinfo->szInputFilename;
    vszModGenName = pinfo->szModGenName;

    sprintf (vszModified_Title, "%s %s", szFileOut, "for R deSolve
package");
    WriteHeader (pfile, vszModified_Title, pinfo->pvmGloVars);

    Write_R_Includes (pfile);
    Write_R_Decls (pfile, pinfo->pvmGloVars);

    Write_R_InitModel (pfile, pinfo->pvmGloVars);
    Write_R_Scale (pfile, pinfo->pvmGloVars, pinfo->pvmScaleEqns);
    Write_R_CalcDeriv (pfile, pinfo->pvmGloVars, pinfo->pvmDynEqns,
                       pinfo->pvmCalcOutEqns); /* fold in CaclOutput */
    Write_R_CalcJacob (pfile, pinfo->pvmGloVars, pinfo->pvmJacobEqns);
    Write_R_Events (pfile, pinfo->pvmGloVars, pinfo->pvmEventEqns);
    Write_R_Roots (pfile, pinfo->pvmGloVars, pinfo->pvmRootEqns);

    fclose (pfile);

    printf ("\n* Created C model file '%s'.\n\n", szFileOut);
}

/* if */
else
    ReportError (NULL, RE_CANNOTOPEN | RE_FATAL,
                 szFileOut, "in Write_R_Model ()");

```

```

/* Write a function to initialize everything */
/* Construct the name of the R file.  If szFileOut is
   fu.c, R file is fu_inits.R
*/
lastdot = strrchr(szFileOut,'.');
if (lastdot != NULL)
    *lastdot = '\0';
nbase = strlen(szFileOut);

/* Length of buffer for new file name: includes terminating null */
nRout = nbase + strlen(Rappend) + 1;
Rfile = (PSTR) malloc(nRout);
Rfile = strncpy(Rfile, szFileOut, nbase);
Rfile[nbase] = '\0';
Rfile = strcat(Rfile, Rappend);
pfile = fopen (Rfile, "w");
if (pfile) {
    Write_R_InitPOS(pfile, pinfo->pvmGloVars, pinfo->pvmScaleEqns);
    fclose(pfile);
    printf ("\n* Created R parameter initialization file
'%s'.\n\n",Rfile);
}
else
    ReportError(NULL, RE_CANNOTOPEN | RE_FATAL,
                Rfile, "in Write_R_Model ()");

free(Rfile);

} /* Write_R_Model */

```